Training a Classifier

Created On: Mar 24, 2017 | Last Updated: Dec 20, 2024 | Last Verified: Not Verified

This is it. You have seen how to define neural networks, compute loss and make updates to the weights of the network.

Now you might be thinking,

What about data?

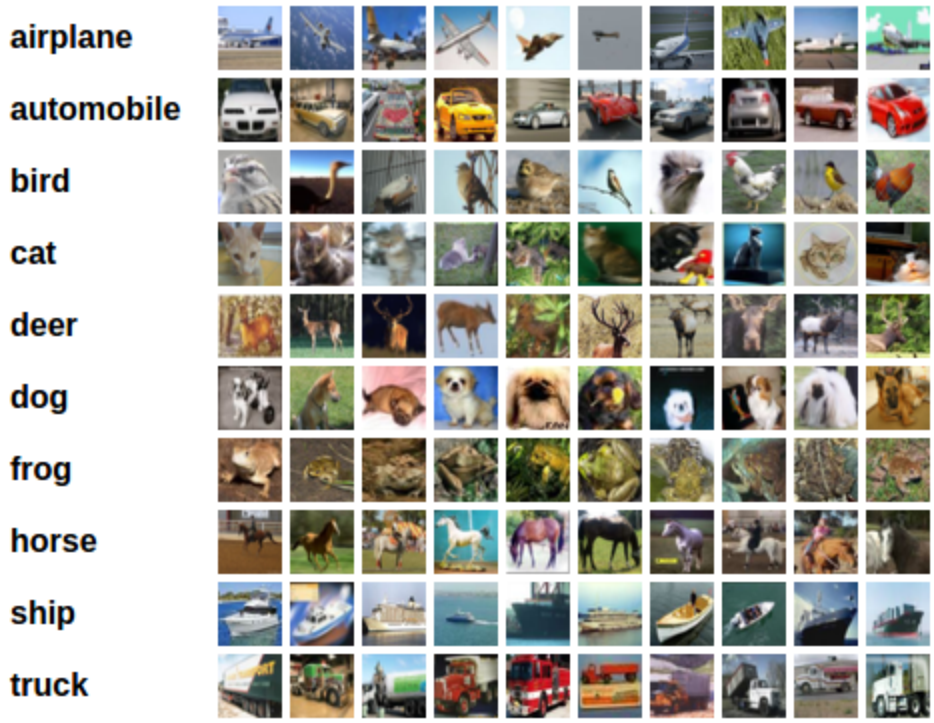
Generally, when you have to deal with image, text, audio or video data, you can use standard python packages that load data into a numpy array. Then you can convert this array into a `torch.*Tensor`.

- For images, packages such as Pillow, OpenCV are useful
- For audio, packages such as scipy and librosa
- For text, either raw Python or Cython based loading, or NLTK and SpaCy are useful

Specifically for vision, we have created a package called `torchvision`, that has data loaders for common datasets such as ImageNet, CIFAR10, MNIST, etc. and data transformers for images, viz., `torchvision.datasets` and `torch.utils.data.DataLoader`.

This provides a huge convenience and avoids writing boilerplate code.

For this tutorial, we will use the CIFAR10 dataset. It has the classes: ‘airplane’, ‘automobile’, ‘bird’, ‘cat’, ‘deer’, ‘dog’, ‘frog’, ‘horse’, ‘ship’, ‘truck’. The images in CIFAR-10 are of size 3x32x32, i.e. 3-channel color images of 32x32 pixels in size.



cifar10

Training an image classifier

We will do the following steps in order:

1. Load and normalize the CIFAR10 training and test datasets using `torchvision`
2. Define a Convolutional Neural Network
3. Define a loss function
4. Train the network on the training data
5. Test the network on the test data

1. Load and normalize CIFAR10

Using `torchvision`, it's extremely easy to load CIFAR10.

```
import torch
import torchvision
import torchvision.transforms as transforms
```

The output of torchvision datasets are PILImage images of range [0, 1]. We transform them to Tensors of normalized range [-1, 1].

Note

If running on Windows and you get a BrokenPipeError, try setting the num_worker of torch.utils.data.DataLoader() to 0.

```
transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

batch_size = 4

trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                        download=True,
transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=batch_size,
                                          shuffle=True, num_workers=2)

testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                       download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=batch_size,
                                         shuffle=False, num_workers=2)

classes = ('plane', 'car', 'bird', 'cat',
           'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
```

```
  0%|          | 0.00/170M [00:00<?, ?B/s]
  0%|          | 426k/170M [00:00<00:41, 4.10MB/s]
  3%|2         | 4.78M/170M [00:00<00:06, 26.9MB/s]
  6%|6         | 10.3M/170M [00:00<00:04, 39.6MB/s]
  8%|8         | 14.4M/170M [00:00<00:03, 40.0MB/s]
 11%|#1        | 19.2M/170M [00:00<00:03, 42.9MB/s]
 14%|#3        | 23.5M/170M [00:00<00:03, 40.1MB/s]
 16%|#6        | 27.6M/170M [00:00<00:03, 38.5MB/s]
 18%|#8        | 31.5M/170M [00:00<00:03, 37.5MB/s]
 21%|##        | 35.3M/170M [00:00<00:03, 36.8MB/s]
 23%|##2       | 39.0M/170M [00:01<00:03, 35.3MB/s]
 25%|##4       | 42.6M/170M [00:01<00:03, 33.1MB/s]
 27%|##6       | 45.9M/170M [00:01<00:03, 32.0MB/s]
 29%|##8       | 49.2M/170M [00:01<00:03, 30.8MB/s]
 31%|###       | 52.3M/170M [00:01<00:03, 30.4MB/s]
 32%|###2      | 55.3M/170M [00:01<00:03, 30.2MB/s]
 34%|###4      | 58.4M/170M [00:01<00:03, 30.2MB/s]
 36%|###6      | 61.4M/170M [00:01<00:03, 30.2MB/s]
 38%|###7      | 64.5M/170M [00:01<00:03, 29.4MB/s]
 40%|###9      | 67.4M/170M [00:02<00:04, 24.7MB/s]
 41%|####1     | 70.1M/170M [00:02<00:04, 21.2MB/s]
 42%|####2     | 72.4M/170M [00:02<00:05, 19.4MB/s]
 44%|####3     | 74.4M/170M [00:02<00:05, 18.1MB/s]
 45%|####4     | 76.3M/170M [00:02<00:05, 17.5MB/s]
 46%|####5     | 78.1M/170M [00:02<00:05, 17.2MB/s]
```

```
 47%|####6     | 79.9M/170M [00:02<00:05, 16.9MB/s]
 48%|####7     | 81.6M/170M [00:03<00:05, 16.9MB/s]
 49%|####8     | 83.3M/170M [00:03<00:05, 16.9MB/s]
 50%|####9     | 85.1M/170M [00:03<00:05, 16.9MB/s]
 51%|#####     | 86.8M/170M [00:03<00:04, 17.0MB/s]
 52%|#####1    | 88.6M/170M [00:03<00:04, 17.2MB/s]
 53%|#####3    | 90.4M/170M [00:03<00:04, 17.3MB/s]
 54%|#####4    | 92.1M/170M [00:03<00:04, 16.5MB/s]
 55%|#####5    | 93.8M/170M [00:03<00:05, 15.3MB/s]
 56%|#####5    | 95.4M/170M [00:03<00:05, 14.6MB/s]
 57%|#####6    | 96.9M/170M [00:04<00:05, 14.0MB/s]
 58%|#####7    | 98.3M/170M [00:04<00:05, 13.8MB/s]
 58%|#####8    | 99.7M/170M [00:04<00:05, 13.6MB/s]
 59%|#####9    | 101M/170M [00:04<00:05, 13.6MB/s]
 60%|######    | 102M/170M [00:04<00:04, 13.7MB/s]
 61%|######    | 104M/170M [00:04<00:05, 12.8MB/s]
 62%|######1   | 105M/170M [00:04<00:05, 11.8MB/s]
 62%|######2   | 106M/170M [00:04<00:05, 11.3MB/s]
 63%|######3   | 108M/170M [00:04<00:05, 11.0MB/s]
 64%|######3   | 109M/170M [00:05<00:05, 10.8MB/s]
 64%|######4   | 110M/170M [00:05<00:05, 10.8MB/s]
 65%|######5   | 111M/170M [00:05<00:05, 10.8MB/s]
 66%|######5   | 112M/170M [00:05<00:05, 10.9MB/s]
 66%|######6   | 113M/170M [00:05<00:05, 10.9MB/s]
 67%|######7   | 114M/170M [00:05<00:05, 11.1MB/s]
 68%|######7   | 115M/170M [00:05<00:04, 11.2MB/s]
 68%|######8   | 117M/170M [00:05<00:04, 10.9MB/s]
 69%|######9   | 118M/170M [00:05<00:05, 9.95MB/s]
 70%|######9   | 119M/170M [00:06<00:05, 9.53MB/s]
 70%|#######   | 120M/170M [00:06<00:05, 9.36MB/s]
 71%|#######   | 121M/170M [00:06<00:05, 9.20MB/s]
 71%|#######1  | 122M/170M [00:06<00:05, 9.01MB/s]
 72%|#######1  | 123M/170M [00:06<00:05, 8.13MB/s]
 72%|#######2  | 123M/170M [00:06<00:06, 7.65MB/s]
 73%|#######2  | 124M/170M [00:06<00:06, 7.46MB/s]
 73%|#######3  | 125M/170M [00:06<00:06, 7.31MB/s]
 74%|#######3  | 126M/170M [00:06<00:06, 6.75MB/s]
 74%|#######4  | 126M/170M [00:07<00:07, 6.17MB/s]
 74%|#######4  | 127M/170M [00:07<00:07, 5.97MB/s]
 75%|#######4  | 128M/170M [00:07<00:07, 5.99MB/s]
 75%|#######5  | 128M/170M [00:07<00:07, 5.96MB/s]
 76%|#######5  | 129M/170M [00:07<00:06, 6.01MB/s]
 76%|#######5  | 129M/170M [00:07<00:07, 5.52MB/s]
 76%|#######6  | 130M/170M [00:07<00:07, 5.14MB/s]
 77%|#######6  | 131M/170M [00:07<00:07, 5.14MB/s]
 77%|#######6  | 131M/170M [00:08<00:07, 5.08MB/s]
 77%|#######7  | 132M/170M [00:08<00:07, 5.11MB/s]
```

```
 78%|#######7  | 132M/170M [00:08<00:07, 5.14MB/s]
 78%|#######7  | 133M/170M [00:08<00:07, 5.26MB/s]
 78%|#######8  | 133M/170M [00:08<00:06, 5.34MB/s]
 79%|#######8  | 134M/170M [00:08<00:06, 5.49MB/s]
 79%|#######8  | 135M/170M [00:08<00:06, 5.68MB/s]
 79%|#######9  | 135M/170M [00:08<00:06, 5.51MB/s]
 80%|#######9  | 136M/170M [00:08<00:06, 5.47MB/s]
 80%|#######9  | 136M/170M [00:08<00:06, 5.22MB/s]
 80%|########  | 137M/170M [00:09<00:06, 5.18MB/s]
 81%|########  | 137M/170M [00:09<00:06, 5.18MB/s]
 81%|########  | 138M/170M [00:09<00:06, 5.19MB/s]
 81%|########1 | 138M/170M [00:09<00:06, 5.28MB/s]
 81%|########1 | 139M/170M [00:09<00:05, 5.35MB/s]
 82%|########1 | 140M/170M [00:09<00:05, 5.49MB/s]
 82%|########2 | 140M/170M [00:09<00:05, 5.54MB/s]
 83%|########2 | 141M/170M [00:09<00:05, 5.73MB/s]
 83%|########2 | 141M/170M [00:09<00:04, 5.87MB/s]
 83%|########3 | 142M/170M [00:09<00:04, 6.05MB/s]
 84%|########3 | 143M/170M [00:10<00:04, 6.28MB/s]
 84%|########4 | 143M/170M [00:10<00:04, 6.45MB/s]
 85%|########4 | 144M/170M [00:10<00:03, 6.65MB/s]
 85%|########4 | 145M/170M [00:10<00:03, 6.80MB/s]
 85%|########5 | 146M/170M [00:10<00:03, 6.99MB/s]
 86%|########5 | 146M/170M [00:10<00:03, 7.11MB/s]
 86%|########6 | 147M/170M [00:10<00:03, 7.31MB/s]
 87%|########6 | 148M/170M [00:10<00:02, 7.51MB/s]
 87%|########7 | 149M/170M [00:10<00:02, 7.69MB/s]
 88%|########7 | 150M/170M [00:10<00:02, 7.90MB/s]
 88%|########8 | 151M/170M [00:11<00:02, 8.07MB/s]
 89%|########8 | 151M/170M [00:11<00:02, 8.14MB/s]
 89%|########9 | 152M/170M [00:11<00:02, 8.33MB/s]
 90%|########9 | 153M/170M [00:11<00:02, 8.48MB/s]
 90%|######### | 154M/170M [00:11<00:01, 8.65MB/s]
 91%|######### | 155M/170M [00:11<00:01, 8.77MB/s]
 91%|#########1| 156M/170M [00:11<00:01, 8.97MB/s]
 92%|#########2| 157M/170M [00:11<00:01, 9.01MB/s]
 93%|#########2| 158M/170M [00:11<00:01, 9.01MB/s]
 93%|#########3| 159M/170M [00:12<00:01, 9.24MB/s]
 94%|#########3| 160M/170M [00:12<00:01, 9.48MB/s]
 94%|#########4| 161M/170M [00:12<00:01, 9.47MB/s]
 95%|#########4| 162M/170M [00:12<00:00, 9.62MB/s]
 96%|#########5| 163M/170M [00:12<00:00, 9.83MB/s]
 96%|#########6| 164M/170M [00:12<00:00, 10.1MB/s]
 97%|#########6| 165M/170M [00:12<00:00, 10.1MB/s]
 97%|#########7| 166M/170M [00:12<00:00, 10.2MB/s]
 98%|#########7| 167M/170M [00:12<00:00, 10.2MB/s]
 99%|#########8| 168M/170M [00:12<00:00, 10.4MB/s]
```

```
 99%|#########9| 169M/170M [00:13<00:00, 10.5MB/s]
100%|#########9| 170M/170M [00:13<00:00, 10.6MB/s]
100%|##########| 170M/170M [00:13<00:00, 13.0MB/s]
```

Let us show some of the training images, for fun.

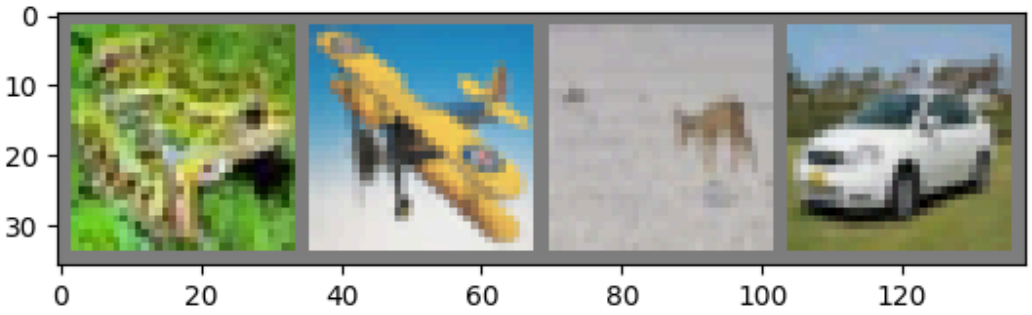
```
import matplotlib.pyplot as plt
import numpy as np

# functions to show an image


def imshow(img):
    img = img / 2 + 0.5     # unnormalize
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.show()


# get some random training images
dataiter = iter(trainloader)
images, labels = next(dataiter)

# show images
imshow(torchvision.utils.make_grid(images))
# print labels
print(' '.join(f'{classes[labels[j]]:5s}' for j in range(batch_size)))
```




```
frog  plane deer  car
```

2. Define a Convolutional Neural Network

Copy the neural network from the Neural Networks section before and modify it to take 3-channel images (instead of 1-channel images as it was defined).

```
import torch.nn as nn
import torch.nn.functional as F


class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)
```

```
    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = torch.flatten(x, 1) # flatten all dimensions except batch
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x


net = Net()
```

3. Define a Loss function and optimizer

Let's use a Classification Cross-Entropy loss and SGD with momentum.

```
import torch.optim as optim

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
```

4. Train the network

This is when things start to get interesting. We simply have to loop over our data iterator, and feed the inputs to the network and optimize.

```
for epoch in range(2):  # loop over the dataset multiple times

    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss += loss.item()
```

```
        if i % 2000 == 1999:    # print every 2000 mini-batches
            print(f'[{epoch + 1}, {i + 1:5d}] loss: {running_loss /
2000:.3f}')
            running_loss = 0.0

print('Finished Training')
```

```
[1,  2000] loss: 2.143
[1,  4000] loss: 1.833
[1,  6000] loss: 1.674
[1,  8000] loss: 1.574
[1, 10000] loss: 1.524
[1, 12000] loss: 1.445
[2,  2000] loss: 1.405
[2,  4000] loss: 1.363
[2,  6000] loss: 1.339
[2,  8000] loss: 1.342
[2, 10000] loss: 1.311
[2, 12000] loss: 1.274
Finished Training
```

Let's quickly save our trained model:

```
PATH = './cifar_net.pth'
torch.save(net.state_dict(), PATH)
```

See here for more details on saving PyTorch models.

5. Test the network on the test data

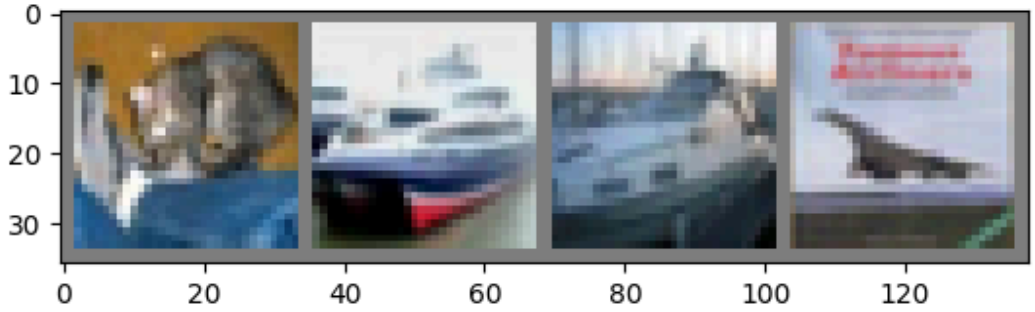
We have trained the network for 2 passes over the training dataset. But we need to check if the network has learnt anything at all.

We will check this by predicting the class label that the neural network outputs, and checking it against the ground-truth. If the prediction is correct, we add the sample to the list of correct predictions.

Okay, first step. Let us display an image from the test set to get familiar.

```
dataiter = iter(testloader)
images, labels = next(dataiter)

# print images
imshow(torchvision.utils.make_grid(images))
print('GroundTruth: ', ' '.join(f'{classes[labels[j]]:5s}' for j in
range(4)))
```




```
GroundTruth:  cat   ship  ship  plane
```

Next, let's load back in our saved model (note: saving and re-loading the model wasn't necessary here, we only did it to illustrate how to do so):

```
net = Net()
net.load_state_dict(torch.load(PATH, weights_only=True))
```

```
<All keys matched successfully>
```

Okay, now let us see what the neural network thinks these examples above are:

```
outputs = net(images)
```

The outputs are energies for the 10 classes. The higher the energy for a class, the more the network thinks that the image is of the particular class. So, let's get the index of the highest energy:

```
_, predicted = torch.max(outputs, 1)

print('Predicted: ', ' '.join(f'{classes[predicted[j]]:5s}'
                              for j in range(4)))
```

```
Predicted:  frog  ship  ship  plane
```

The results seem pretty good.

Let us look at how the network performs on the whole dataset.

```
correct = 0
total = 0
# since we're not training, we don't need to calculate the gradients for
our outputs
with torch.no_grad():
    for data in testloader:
        images, labels = data
        # calculate outputs by running images through the network
        outputs = net(images)
        # the class with the highest energy is what we choose as
prediction
        _, predicted = torch.max(outputs, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print(f'Accuracy of the network on the 10000 test images: {100 * correct
// total} %')
```

```
Accuracy of the network on the 10000 test images: 55 %
```

That looks way better than chance, which is 10% accuracy (randomly picking a class out of 10 classes). Seems like the network learnt something.

Hmmm, what are the classes that performed well, and the classes that did not perform well:

```
# prepare to count predictions for each class
correct_pred = {classname: 0 for classname in classes}
total_pred = {classname: 0 for classname in classes}

# again no gradients needed
with torch.no_grad():
    for data in testloader:
        images, labels = data
```

```
        outputs = net(images)
        _, predictions = torch.max(outputs, 1)
        # collect the correct predictions for each class
        for label, prediction in zip(labels, predictions):
            if label == prediction:
                correct_pred[classes[label]] += 1
            total_pred[classes[label]] += 1


# print accuracy for each class
for classname, correct_count in correct_pred.items():
    accuracy = 100 * float(correct_count) / total_pred[classname]
    print(f'Accuracy for class: {classname:5s} is {accuracy:.1f} %')
```

```
Accuracy for class: plane is 43.5 %
Accuracy for class: car   is 67.2 %
Accuracy for class: bird  is 44.7 %
Accuracy for class: cat   is 29.5 %
Accuracy for class: deer  is 48.9 %
Accuracy for class: dog   is 47.0 %
Accuracy for class: frog  is 60.5 %
Accuracy for class: horse is 70.9 %
Accuracy for class: ship  is 79.8 %
Accuracy for class: truck is 65.4 %
```

Okay, so what next?

How do we run these neural networks on the GPU?

Training on GPU

Just like how you transfer a Tensor onto the GPU, you transfer the neural net onto the GPU.

Let's first define our device as the first visible cuda device if we have CUDA available:

```
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')

# Assuming that we are on a CUDA machine, this should print a CUDA device:

print(device)
```

```
cuda:0
```

The rest of this section assumes that `device` is a CUDA device.

Then these methods will recursively go over all modules and convert their parameters and buffers to CUDA tensors:

```
net.to(device)
```

Remember that you will have to send the inputs and targets at every step to the GPU too:

```
inputs, labels = data[0].to(device), data[1].to(device)
```

Why don't I notice MASSIVE speedup compared to CPU? Because your network is really small.

Exercise: Try increasing the width of your network (argument 2 of the first `nn.Conv2d`, and argument 1 of the second `nn.Conv2d` – they need to be the same number), see what kind of speedup you get.

Goals achieved:

- Understanding PyTorch's Tensor library and neural networks at a high level.
- Train a small neural network to classify images